**Amendments to the Specification:**

Please replace the paragraph on Page 2, lines 1-13, with the following amended paragraph:

The situation described above is not unique to surface modeling. Software engineering discipline faces similar problems. When modeling software engineering artifacts, a user can often be frustrated because control points provided to the user are closely tied to the representations' degree of freedom. In this vein, one may consider object-oriented (OO) design methodology for software development. An OO language, such as JAVA language or C++, provides a user with a few control points for manipulating software artifacts. Inheritance lets a user extend a class. In that class extensions tend not to represent the only type of controls that a user wants, it is often very difficult, if not impossible, to make a simple change, such as adding logging feature to a set of classes. [14]. Accordingly, a need has been recognized in connection with providing a user with an infinitely malleable software artifact having no fixed structure or controls of its own. To this software artifact additional features or control points can then be added to obtain concrete controllable software artifacts.

Please replace the paragraph on Page 6, lines 9-11, with the following amended paragraph:

~~HyperJ~~ HYPERJ extension framework (see below) provides a simpler solution for Decorator pattern using a composition rule similar to the following:

override(Display, (DecoratedDisplay, Display))

Please replace the paragraph bridging Page 6, line 12-page 7, line 2, with the following amended paragraph:

This essentially merges the two classes (the old Display and DecoratedDisplay) into one class. The new class is still called Display and it contains all functionality of the two classes. A method such as paint() will be picked up from DecoratedDisplay in the new Display class. The ~~HyperJ~~ HYPERJ extension framework approach is completely non-invasive; it does not touch any of the client code. But a disadvantage of ~~HyperJ~~ HYPERJ extension framework approach is that every client will get the new Display. One way to deal with this is to use a new name for Display:

override(NewDisplay, (DecoratedDisplay, Display))

Please replace the paragraph bridging Page 7, line 12-page 8, line 4, with the following amended paragraph:

The DecoratedDisplay class contains the enhanced paint method, and so a call to ed.paint() will invoke the new *paint()* method, overriding the old method paint() in Display. Now if one wants to invoke the old paint() method, then on can do so in an extension type using the syntax ed(1).paint(). Extension types are ordered tuples, and so one can access elements of a tuple via indexing. The extension type

(DecoratedDisplay,Display) is a sub-type of both Display and DecoratedDisplay types, and therefore can masquerade the two element types. Unlike the ~~HyperJ~~ HYPERJ extension framework approach, one does not create any new classes. Also, the extension type composition is at instance-level rather than at class-level. Therefore, users that do not care about the DecoratedDisplay need not extend their Display object.

Please replace the paragraph on Page 8, lines 5-17, with the following amended paragraph:

Herein there is also described the semantics of extension types, and several applications of extension types are given in the context of aspect-oriented software development and design patterns. There is introduced the notion of parameterized extension types that provide more type-safe composition than is possible with pure extension types. There are also presented manners of modeling both multiple classification and dynamic classification using extension types (see [9]). (Particularly, whereas [9] is an early article showing multiple/dynamic classification, it hard-codes the methodology rather than use a more flexible system as contemplated herein). Finally, it will be shown how extension types can added to an ASPECTJ extension framework to provide a notion sub-typing to classes and aspects (see [6]). (Particularly, [6] represents an early paper introducing ASPECTJ extension framework. While aspects in ASPECTJ extension framework are not first class types, it will be demonstrated herein that by using extension types, aspects can be configured as first class types.)

Please replace the paragraph on Page 9, lines 12-18, with the following amended paragraph:

Separation of concerns (SOC), a term introduced by Djisktra, refers to a software engineering concept for identifying, encapsulating, and manipulating different features (aspects, concerns, etc) of software artifacts so that one can organize and decompose software into manageable and comprehensible parts. [13][14] A class in OO languages is one kind of a concern. There are others kinds of concern that can cut across multiple classes, such as logging, printing, persistence, and display capabilities. HYPERJ and ASPECTJ extension frameworks are two different implementations of SOC in JAVA language [13][6].

Please replace the paragraph bridging Pages 10-11 with the following amended paragraph:

HYPERJ extension framework uses concepts such as hyperslice and hyperspace. [13] A hyperspace is a "space of concerns" that is spanned by a set of "vectors of concerns". A set of "vector of concerns" is called a hyperslice of a hyperspace. A "basis concern" for a hyperspace is an independent set of concerns that span the hyperspace. A dimension of a hyperspace *H* is the number of elements (concerns) in a basis concern for *H*. A hypermodule *M* is a sub-hyperspace of a hyperspace *H* that consists of hyperslices along with operations for composing hyperslices. Hypermodules are building blocks, and

are not, in general, complete, executable programs. A system is a hypermodule that is complete, and can therefore run independently. OO languages such as JAVA language and C++ support what is termed as a "tyranny of dominant decomposition," or separation and encapsulation along only one dominant hyperslice. [13][14]. For instance, consider the class hierarchy of employees in a company as shown in Fig. 1. As shown, the class hierarchy contains at least two hyperslices that are tangled: (1) personnel hyperslice including employee name, identity, management hierarchy, etc., and (2) payroll hyperslice including salary, tax, and other related information. There is a poor separation of concerns in such a class design. One way to separate the personnel concern from payroll concern is to create two class hierarchies: one for personnel department and another for payroll department. An Employee class hierarchy can then be constructed by appropriately composing the two hyperslices. HYPERJ extension framework, for instance, proposes a set of composition operations on hyperslices. An interesting aspect of the HYPERJ extension framework approach is that composition operations are separate from hyperslices. This allows one to construct new class hierarchies in a non-intrusive manner.

Please replace the paragraph on page 11, lines 4-13, with the following amended paragraph:

Accordingly, let $H_1, H_2, \ldots, H_n$ be a set of hyperslices. Let *CH* be a new class hierarchy that was constructed by composing $H_1, H_2, \ldots, H_n$ using a set of composition

operations *O*. With this in mind, a disadvantage of the ~~HyperJ~~ HYPERJ extension framework approach is that a new class hierarchy is created for every new set of operations on $H_1$, $H_2$, ..., $H_n$; the new class hierarchy *CH* has no semantic relation to $H_1$, $H_2$, ..., $H_n$. For instance, a class in *CH* need not be a sub-type of a class in any $H_1$, $H_2$, ..., $H_n$. In ~~HyperJ~~ HYPERJ extension framwork, hyperslice composition happens at compilation/loading time, creating new classes. ~~HyperJ~~ HYPERJ extension framwork does not support the notion of "instance" or "object" level composition of hyperslices. Due to these limitations ~~HyperJ~~ HYPERJ extension framwork does not support dynamic variational modeling, such as, for instance, a research employee decides to become a sales executive.

Please replace the paragraph bridging Pages 11-12 with the following amended paragraph:

ASPECTJ extension framework is a general-purpose aspect-oriented programming (AOP) extension to JAVA language. [6] In ~~AspectJ~~ ASPECTJ extension framework, "aspects" modularize concerns that affect one or more classes. ASPECTJ extension framework uses the notion of "join points" to insert new behavior in existing code. Join points are well-defined points in the execution of a program, which includes, reading or writing a field; calling or executing an exception handler, method or constructor. These join points are described by the "pointcut" declaration. Pointcuts are typically defined in aspects. An "advice" is a piece of code that is executed at each join

point defined in a pointcut. There are three kinds of advice: before advice, around advice and after advice. Pointcuts and advice are encapsulated in an aspect. An "aspect" is like a class that encapsulates pointcuts and advices. An aspect can also include methods and field and can extend another class or aspect.

Please replace the paragraph on Page 12, lines 7-14, with the following amended paragraph:

One cannot create instances of aspects or type program variables as aspect types in ASPECTJ extension framework. In other words, aspects are not first-class types in ASPECTJ extension framework. Also, there is no notion of sub-type relation among aspects, although one aspect can extend another aspect. The semantics of aspect inheritance is not same as for classes. When one aspect extends another aspect it does not override pointcuts or advices defined in the parent aspect. In ASPECTJ extension framework, first the child pointcut is executed and then the parent pointcut is executed. For instance, one may consider the Observer pattern (see also *supra*). [4] The following is a code snippet of the Notify aspect:

Please replace the paragraph bridging Pages 13-14 with the following amended paragraph:

Both the ASPECTJ extension framework and HYPERJ extension framework approaches go beyond traditional OO concepts. The basic premise for both approaches is that modularity goes well beyond what can be expressed in traditional OO languages. Similar patterns of code fragments occur in many places (e.g. logging code fragment). ASPECTJ extension framework achieves modularity by collecting all similar patterns into a single *aspect*, and provides rules for injecting those patterns into other classes as and when needed. Code injection happens at compilation time. HYPERJ extension framework also follows a similar approach, but treats similar code patterns as a separate hyperslice and provides rules for composing hyperslices. Once again, hyperslice composition happens at compilation time. In order to make ASPECTJ extension framework more fluid or HYPERJ extension framework more morphogenic, one needs to go beyond compile time compositions. [22] One needs to treat aspects in ASPECTJ extension framework as first-class types, as well as hyperslice in HYPERJ extension framework. Once one treats aspects and hyperslices as first-class types, one can then compose them dynamically as needed. Also, once one treats them as first-class types one can apply standard type analysis to ensure type-safe composition. Extension types are one way to achieve the above goals. In extension types , one can treat hyperslices and aspects as first-class types and by treating them as first-class types one increases the expressiveness of both ASPECTJ and HYPERJ extension frameworks approaches.

Please replace the paragraph on Page 15, lines 6-18 with the following amended paragraph:

Class-based languages, such as JAVA language and C++, provide a one-dimensional view of a class hierarchy; that is, two classes are related (through sub-class relation) only if one of them is an ancestor of the other in the class hierarchy. The control point (i.e., the sub-type relation) is closely tied to the representation (i.e., the class hierarchy). The close tie between sub-type relation and class hierarchy has been debated elsewhere in other contexts (this will be treated later). HYPERJ extension framework goes one step further and views a class hierarchy as one hyperslice and provides composition operations to compose several such hyperslices to create a new class hierarchy that somehow captures all the features of the individual hyperslices. What is missing in HYPERJ extension framework is the semantic relation (i.e., control points) between the new class and the old hyperslices. Rather than explicitly create a new class for every set of composition operation and (possibly) discard the old hyperslices, a new composed type (extension type) is now contemplated which establishes a sub-type relation between the new type and its constituent types.

Please replace the paragraph on Page 17, lines 9-14 with the following amended paragraph:

With regard to method dispatch, let $P$ be the declared type of $p$ and $Q$ be the run-time type of an object $o$ pointed by $p$. A method lookup $p.m()$ in JAVA language involves

walking up the class hierarchy from $Q$ and finding the closest implementation of $m$. In an extension type one can define different kinds of method dispatches. Let $\alpha$ be the extension type of a variable $p$ and let $\beta$ be the runtime extension type of the object pointed by $p$, so that $\beta <: \alpha$. With this in mind:

Please replace the paragraph bridging pages 17-18 with the following amended paragraph:

**1.** ***p.m*** method dispatch is very similar to the traditional dispatch, except that one may look for $m$ in all element class hierarchies. One preferably starts at the element type $\beta(0)$ and walks up the class hierarchy of $\beta(0)$ to find the closest $m$. If $m$ is not defined in the class hierarchy of $\beta(0)$, one then looks for $m$ in $\beta(1)$ class hierarchy. This process is repeated for each $|\beta|$ class hierarchies of $\beta$ and find the first $m$ and dispatch the method $m$. Notice that the method $m$ should be declared in at least one element class of $\alpha$. The method dispatch *p.m()* is like the "override" rule in ~~HyperJ~~ HYPERJ extension framework.

Please replace the paragraph on Page 18, lines 5-10 with the following amended paragraph:

**2.** ***p*m*** method dispatch is defined as follows: For each element type $\beta(i)$, in the order $i=0, \ldots, |\beta|-1$, walk up the class hierarchy of $\beta(i)$ to find the closest $m$ in $\updownarrow(i)$ and

dispatch the method *m* (if found). It is a type error if *m* is not defined in at least one of $\updownarrow(i)$, $i=0, \ldots, |\beta|-1$, class hierarchy. Notice that in this case one may invoke at most $|\beta|$ methods. The method *m* should be declared in at least one element class of $\alpha$. The method dispatch *p*m()* is motivated by the correspondence rule "merge-by-name" in ~~HyperJ~~ HYPERJ extension framework.

Please replace the paragraph on Page 19, lines 10-15 with the following amended paragraph:

The method dispatch p.m() is like the override rule in HyperJ. The method dispatch p*m() is motivated by correspondence rule "merge-by-name" in ~~HyperJ~~ HYPERJ extension framework. This rule allows one to combine methods from different hyperslices into one method in the new composed hyperslice. A call to the new method will in turn invoke each of the old method. One can define other kinds of dispatch mechanism that can handle other kinds of ~~HyperJ~~ HYPERJ extension framework composition rules.

Please replace the paragraph bridging pages 21-22 with the following amended paragraph:

~~HyperJ~~ HYPERJ extension framework provides several kinds of composition rules. Herein there are only discussed "override" and "merge-by-name" correspondence

rules. It is possible to encode other rules at the time when extension objects are created. For instance, consider the following two classes:

```
class A {... ; void m() {...} ...}

class B {... ; void n() {...} ...}
```

One may create an extension type

```
typedef (A,B) p ;
```

Please replace the paragraph on Page 22, lines 7-9 with the following amended paragraph:

Now assume that one wants to ensure that method *m()* and *n()* are same, that is, the "equate" rule in ~~HyperJ~~ HYPERJ extension framework. One can do this at the time when extension object is created.

```
p = new (A,B) {equate A.m, B.n} ;
```

Please replace the paragraph on Page 22, lines 10-11 with the following amended paragraph:

A method dispatch p.n() is then the same as p.m(). (There has not yet been explored other rules in ~~HyperJ~~ HYPERJ extension framework that can be encoded in extension types.)

Please replace the paragraph on Page 22, lines 13-14 with the following amended paragraph:

The disclosure now turns to how extension types can be used in AOSD. First, the HYPERJ extension framework approach will be focused upon, and thence the ASPECTJ extension framework approach.

Please replace the paragraph bridging pages 24 and 25 with the following amended paragraph:

Now one can invoke the method emp.benefits() to determine the benefits of an employee. The business rules of personnel department are in a separate class hierarchy from that of the business rules of the payroll department. The two class hierarchies, personnel and payroll, can evolve independently without impacting others. Unlike ~~HyperJ~~

HYPERJ extension framework, one need not create any new (composed) class hierarchy, and yet the benefits of separation of concern are still obtained. In ~~HyperJ~~ HYPERJ extension framework one may create a large number of new classes to accommodate different combinations of compositions.

Please replace the paragraph on Page 25, lines 10-14, with the following amended paragraph:

Turning to ASPECTJ extension framework and extension types, ASPECTJ extension framework (as described in [6]) supports what Kiczales refers to as static AOP [22]. In [6], the aspects and classes are relatively fixed or static in that changing them involves editing the program. Fluid AOP will allow the easy remodularization of both aspects and classes. This dynamic remodularization is related to dynamic classification.

Please replace the paragraph on Page 26, lines 7-8, with the following amended paragraph:

Now one can extend the Notify aspect (similar to what is done in ASPECTJ extension framework) to create a new aspect:

Please replace the paragraph on Page 27, lines 1-4, with the following amended paragraph:

One may note that the extension type (Employee, Notify) contains both Employee class and Notify aspects. In the context of ASPECTJ extension framework one can define an "aspect extension type" to include both classes and aspects as elements. In general, an aspect extension type can be defined as follows:

Please replace the paragraph bridging pages 28-29, with the following amended paragraph:

The construct Personnel <p <: Employee> indicates that the type Personnel can be an element of an extension type whose principal element is a subtype of Employee. A sub-type of Personnel can further restrict p. For instance, the Manager type is restricted to only Regular types. When constructing an extension type, the type system will perform appropriate type checking taking into account the restrictions on the parameters. For example,

(Regular, Manager) v ;

is a valid extension type. On the other hand

(Employee, Manager) iv ;

is not a valid extension, since the Manager feature cannot be applied to all employees but only to a sub-type of Regular employees. It is to be noted that the way the type parameter p is used to further restrict sub-type relation is similar to virtual typing. [11] The concept of principal element type hierarchy is related to principal dimension or hyperslice in ~~HyperJ~~ HYPERJ extension framework. [13]

Please replace the paragraph bridging Pages 29-30 with the following amended paragraph:

A Visitor pattern allows one to add a new operation to a class hierarchy without modifying the classes in the class hierarchy. Traditionally, in languages like JAVA language, a Visitor pattern is implemented using a double-dispatch mechanism. Using HYPERJ or ASPECTJ extension framworks, one can avoid such a double-dispatch mechanism. Herebelow, it will be shown that by using parameterized extension type one can implement a type-safe Visitor pattern using a single dispatch mechanism.

Please replace the paragraph on Page 35, lines 1-7, with the following amended paragraph:

Consider the Observer pattern discussed earlier. In languages like JAVA language that do not support multiple inheritance of classes, Employee will be a sub-class (i.e., sub-type) of the class Subject. But conceptually it is known that Employee is

not a Subject. In other words, there is no conceptual relation between the Employee class and the Subject class. The Employee class really does not care about attach() and detach() method, and it only requires the notify() method (not even how the notify() method is actually implemented).

Please replace the paragraph bridging Pages 38-39, with the following amended paragraph:

Harrison and Ossher introduce *group*-object where a set of objects have single identity. [18]. A group-object is created by composition of primitive objects, and a group-object simply calls methods of primitive objects as defined by the composition operation. The composition operations are similar to those used in ~~HyperJ~~ HYPERJ extension framework, but performed at object level. A group-object behaves like a method combination dispatcher determining how the composed behavior of each method call is to be realized in terms of the primitive methods supplied by the group members. Group-object design was motivated by a need to model collaboration of objects rather than create a single kind of objects as is done in ~~HyperJ~~ HYPERJ extension framework (and other AOSD technique). Harrison and Ossher focus on call dispatch mechanism to model object collaboration. Harrison and Ossher do not provide any typing relation between a group-object and its primitive objects. An extension type is a kind of group-object, but our focus is on composing at type-level and also on modeling type-level AOSD. Also, our method dispatch semantics is different from that of group-object

dispatch mechanism. Lea also proposes a mechanism for grouping related set of objects. [24]. He also focuses on method dispatch mechanism and proposes an alternative channel-based dispatch mechanism.

Please replace the paragraph on Page 39, lines 11-19, with the following amended paragraph:

Mezini and Ostermann use concepts of family polymorphism in Caesar. [20]. Their main goal is to extend ASPECTJ extension framework approach to allow a more flexible reuse and componentization of aspects. Family polymorphism allows one to group a set of classes to participate in collaboration. [21]. Interestingly one can use virtual types to provide a type safe family extension. Extension types are related to family polymorphism; the set of elements in an extension type belong to the same family type. Unlike Caesar or other similar approaches one can mix and match elements to create family types on-the-fly. On the other hand family classes are statically created in Caesar. Also, the semantics of Caesar family class is quite different compared to the semantics of extension types.

Please replace the paragraph on Page 40, lines 14-18, with the following amended paragraph:

In recapitulation, there have been introduced herein extension types for variational modeling. It has been demonstrated how extension types help simplify many concepts in AOSD and design patterns. Extension types is a simple way to implement multiple and dynamic classifications. Future steps could involve providing static and dynamic semantics for extension types, and to implement extension types in JAVA language.